Fundamentals of Data Structures & Algorithms

Assignment 1 – Infix & Postfix Expressions

Wtite a Java program that takes as input a String that represents an expression given in infix notation and first converts it to a String representing the equivalent expression given in postfix notation and then evaluates the converted postfix expression.

Your program can assume the input String is valid and consists of a series of operands and operators (including parentheses) seperated by a space character. Details of the algorithms required and exapmles are provided below.

Your program should make use of appropriate methods to help organise your code and evidence of testing should be provided.

Infix & Postfix Expressions

Infix notation is the notation commonly used in arithmetical and logical formulae and statements. In an infix expression operators are placed between operands, such as the plus sign in 2 + 2. Parentheses surrounding groups of operands and operators are necessary to indicate the intended order in which operations are to be performed. In the absence of parentheses, certain precedence rules determine the order of operations.

Reverse Polish notation (RPN), or postfix notation, is a mathematical notation in which operators follow their operands, the infix expression above would be written as 2 2 +. It does not need any parentheses as long as each operator has a fixed number of operands.

Converting Infix to Postfix

Edsger W. Dijkstra invented the shunting-yard algorithm to convert infix expressions to postfix expressions, so named because its operation resembles that of a railroad shunting yard.

A simplified version of this algorithm is:

while characters remain in the infix string

1. read the next character in the infix string
2. if the character is an operand, append the character to the postfix expression
3. if the character is an operator @
   - while the stack is not empty and an operator of greater or equal priority is on the stack
     - pop the stack and append the operator to the postfix
   - push @
4. if the character is a left parenthesis (
   - push the parenthesis onto the stack
5. if the character is a right parenthesis)
   - while the top of the stack is not a matching left parenthesis (
     - pop the stack and append the operator to the postfix expression
   - pop the stack and discard the returned left parenthesis
6. Pop any remaining items on the stack and append to the postfix expression

Examples

Input	Stack	Postfix
2*3 + 4*5	empty	
*3+4*5	empty	2
3+4*5	*	2
+4*5	*	23
4*5	+	23*
*5	+	23*4
5	*+	23*4
	*+	23*45
	+	23*45*
	empty	23*45*+

Input	Stack	Postfix
(2-3+4)*(5+6*7)	empty	
2-3+4)*(5+6*7)	(	
-3+4)*(5+6*7)	(	2
3+4)*(5+6*7)	(-	2
+4)*(5+6*7)	(-	23
4)*(5+6*7)	(+	23-
)*(5+6*7)	(+	23-4
*(5+6*7)	empty	23-4+
(5+6*7)	*	23-4+
5+6*7)	(*	23-4+
+6*7)	(*	23-4+5
6*7)	+(*	23-4+5
*7)	+(*	23-4+56
7)	*+(*	23-4+56
)	*+(*	23-4+567
	*	23-4+567*+
	empty	23-4+567*+*

Evaluating Postfix Expressions

To evaluate postfix expressions, the postix expression is read left to right. Operands are pushed onto a stack. Whenever and operator is encountered, a fixed number of operands are popped from the stack, the operator is applied to the operands and the result pushed back onto the stack. At the end of input, the result of the expression will be on the stack.

Wriiten as an algorithm, this becomes:

while characters remain in the postfix string

1. 1ead a character
2. if the character is a digit, push to the stack
3. if the character is an operator
   pop the stack twice obtaining the two operands
   perform the operation
   push the result back onto the stack
4. Pop the final value from the stack.

Examples

Input	Stack (top is on the left)	
2 3 4 * +	empty	push 2
3 4 * +	2	push 3
4 * +	3 2	push 4
* +	4 3 2	pop 4, pop 3, do 3 *4 , push 12
+	12 2	pop 12, pop 2, do 2 + 12, push 14
	14	

Input	Stack	
3 4 * 2 5 * +	empty	push 3
4 * 2 5 * +	3	push 4
* 2 5 * +	4 3	pop 4, pop 3, do 3*4, Push 12
2 5 * +	12	push 2
5 * +	2 12	push 5
* +	5 2 12	pop 5, pop 2, do 2*5, push 10
+	10 12	pop 10, pop 12 do 12 + 10, push 22
	22	